I would like to thank Dr. James Burnside, Mr. Glenn Wiggins, Mr. Alan Klick, and my mother, for without their help this work would not have been possible.

TABLE OF CONTENTS

PREFACE

This manual and accompaning assembly language simulator is to be used as a teaching tool in a beginning assembly language course. It will help the student grasp the concepts behind machine-oriented commands, stack functions, branching procedures, and addressing techniques.

Because of the purpose that the simulator serves, simplicity was stressed over actual overall usefulness of the language in a business type environment. The number of operators have been decreased compared to other assemblers. Also, input-output control is limited. However, these shortcomings are outweighed in an academic environment by the simplicity and speed by which the simulator is used. The student still has access to conditional and unconditional branching, subroutines, stack manipulation, alphanumeric variable and label addressing, simple integer and label addressing, simple integer and character input/output, and arrays.

The text is written in a reference manual fashion, rather than a textbook fashion. Some attempt has been made to explain the underlying concepts behind assembly language programming and addressing techniques; however, it is recommended that the user either be familiar with some form

of assembly language already or be under an instructor's supervision before trying to understand the concepts presented here.

6502 ASSEMBLY LANGUAGE SIMULATOR REFERENCE MANUAL

PART I: THE SIMULATOR

This manual serves as an introduction to programming in assembly language using modified 6502 microprocessor code and the assembly language simulator programmed for use on the HP-3000. The language includes 49 mnemonics, including 6 mnemonics not included in the 6502 instruction set (CCT, PRT, PRA, INP, INA, and STP). The simulator can accept implied, immediate, absolute indexed, indirect, variable, or labeled addressing techniques. These will be discussed in detail in Part II.

The system is set up with limited memory space; however, the simulator is easily modified to accommodate larger storage for variables, labels, and memory locations.

MEMORY LOCATIONS

Memory locations available to the programmer include the accumulator, the X- and Y-registers, the program stack, the status register, and the program counter.

THE ACCUMULATOR

The accumulator is nothing more than an eight-bit word stored in main memory. However, many operations depend on the accumulator, especially branch statements and logical

operations. Also, the status register is set based on what is stored in the accumulator at the time. The accumulator can be loaded from memory (LDA), stored in to another location (STA), be added to (ADC) or subtracted from (SBC), or be operated on by logical operators (AND, EOR, ORA). The uses of the accumulator will be discussed throughout the text.

THE X- AND Y-REGISTERS

The X- and Y-registers are locations similar to the accumulator; however, their functions are more limited. They can be loaded from memory (LDX, LDY), stored into memory (STX, STY), or they can be incremented or decremented (INX, INY, DEX, DEY). These registers are also used in X- and Y-indexed addressing. Indexed addressing is described in Part II.

THE STACK

The program stack is an array of 100* bytes in main memory, which can be stored into, changed, or output as desired by the programmer. The zero position in the stack is the accumulator. Therefore, a call to the zero position in the stack is the same as a call to the accumulator. If any variables are defined in a program, they are assigned memory locations starting at position 1 in the stack (see Figure 1).

* can be expanded

There are commands which "push" or "pop" data from the top of the stack. At the beginning of the program the top of the stack is located at position 100*. Any call to push data onto the stack would result in the number being stored at the top of the stack, the stack pointer being decremented by one, and the number being pulled from the top of the stack (Figure 2).

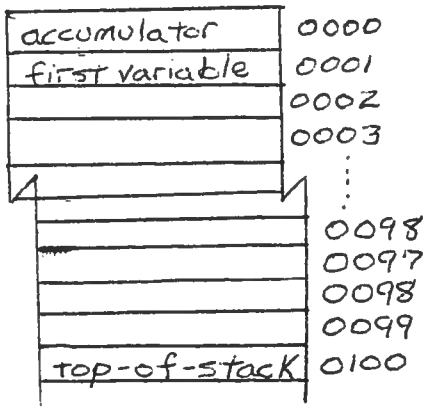

Figure 1: The Program Stack

The stack limit is 100*, therefore the total number of variables in a particular program plus the total number of locations used by the stack must not exceed 100*. Also calls to subroutines use the stack, so leaving any pending calls to the stack when returning from a subroutine or trying to pop a value from a subroutine that is pushed from another procedure will result in an error.

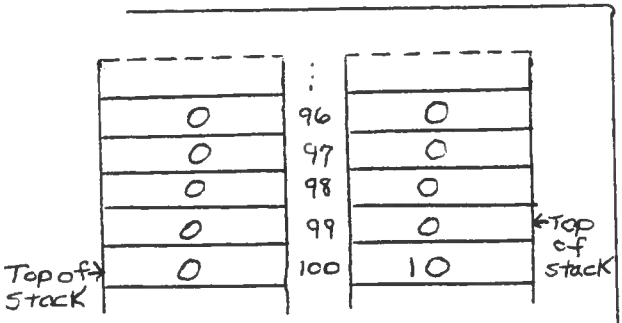

Figure 2: Results of pushing a 10 onto the top of the stack.

Direct calls can be made from the stack. (For instance, INP $0023 would input a number from a terminal or other input device and store it in the 23rd position in the program stack.) However, care must be taken to insure that the memory location

has not already been used as a variable or as a location pushed onto the stack.

THE STATUS REGISTER

The status register consists of "flags" which tell the programmer the status of the accumulator at any particular time. These flags are used in all conditional branching. The flags used by the programmer are the overflow flag (V), the positive flag (P), the zero flag (Z), and the negative flag (N).

The Overflow Flag

Every time the accumulator is changed to a value greater than 255 or less than -255, the accumulator is "rolled over" and the overflow flag is changed. Figure 3 shows the effect of addition and subtraction on the accumulator and overflow flags. Notice that if the overflow flag is already set, a second overflow will clear it. Therefore, it may be a good idea in some applications to clear the overflow flag (CLV) immediately after it is found to be set. The CLV command sets the overflow flag to zero. The BVC command branches on V=0. The BVS command branches on V=1.

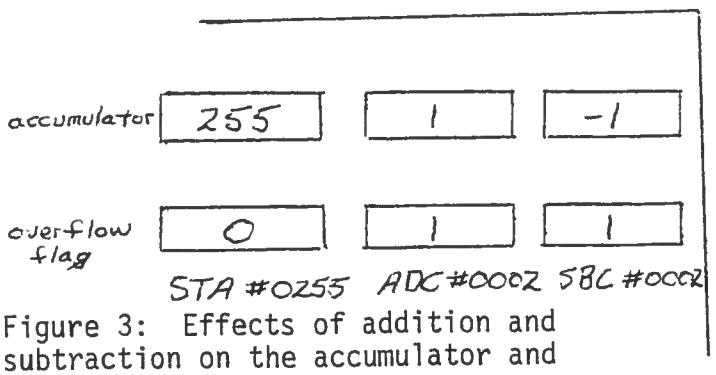

Figure 3: Effects of addition and subtraction on the accumulator and overflow flags.

The Positive Flag

The positive flag is set whenever the value is the accumulator is greater than zero. The BPL command branches on P=1.

The Zero Flag

The zero flag is set whenever the value in the accumulator is equal to zero. The BEQ command branches on Z=1. The BNE command branches on Z=0.

The Negative Flag

The negative flag is set whenever the value in the accumulator is less than zero. The BMI command branches on N=1.

THE PROGRAM COUNTER

The program counter stores the value of the line number that the program is currently executing. The line number of the first executable line in a program (any line other than LAB or VAR) has a line number of 1. All other executable lines of code are numbered relative to that. (Line numbers are printed on the source listing as an aid to debugging).

The program counter is changed by conditional or unconditional branching. It is also changed by calling a subroutine, with the added feature that the original value

of the counter is stored in the stack to be retrieved when returning from that subroutine.

LABELS AND VARIABLES

Labels and variables are used as aids in programming. They enable the programmer to substitute alphanumeric character strings in place of numbers when referencing memory locations or line numbers. Labels and variables can be of any length, but the system only recognizes up to the first ten characters as significant.

VARIABLES

Variables can be declared in place of memory location numbers on the program stack. Variables are assigned starting at location $0001 on the stack. For instance, declaring V1, V2, and V3 to be variables would assign locations $0001, $0002, and $0003 on the program stack to those variables, respectively. So a call to V1 would be the same as a call to location $0001 on the stack, V2 the same as $0002, and V3 the same as $0003.

LABELS

Labels identify line numbers in a program. Labels must be declared before they can be used. If a statement is labeled, then any branch to that label is the same as a branch to the corresponding line number. For instance, if

line number 14 has a label of "LABEL1" then any reference to LABEL1 is the same as a reference to line number 14. Labeled addressing techniques are discussed in Part II.

DECLARING VARIABLES AND LABELS

Variables and labels must be declared before they can be used. Labels are declared using the LAB command, while variables are declared using the VAR command. LAB and VAR commands are non-executable statements, therefore, they do not receive a line number, nor are they part of the object module. The only purpose that the VAR and LAB statements serve is to set up a table in main memory to keep track of variables and label locations. Variables and labels should be declared as soon as possible to avoid trying to reference a variable or label before it is declared.

PART II: SYNTAX AND ADDRESSING TECHNIQUES

SYNTAX

Every line of a program consists of an operator, an addressing technique, and an operand. Optional elements include labels and comments.

GENERAL SYNTAX REQUIREMENTS

A line must be no more than 72 characters long. It can start anywhere on the line, but must not run onto the next line. Blank lines are not allowed. Here is the general form of a line:

[LABEL:] OPERATOR TECHNIQUE OPERAND [COMMENT]

LABEL (Optional)- A label can contain any alphanumeric characters, except colons and spaces. It must be followed immediately by a colon. There may be any number of spaces following the colon.

OPERATOR- An operator consists of a three letter mnemonic which the system can understand. A list of operators are included in the next section. The operator must be followed by one or more spaces.

ADDRESSING TECHNIQUE- A one character code which the system recognizes. Specific addressing techniques are discussed later in this section. A space or spaces after the addressing technique character are optional.

OPERAND- May be a four digit number, a variable, or a label, depending on the addressing technique. The operand must be followed by at least one space.

COMMENT (Optional)- After reading the space after the last character in the operand, the system ignores any other data on the line. Therefore, comments can be inserted up to the 72 character limit (end of line).

ADDRESSING TECHNIQUES

Addressing techniques allow the programmer the flexibility to access data in many different ways. The addressing techniques offered by the system include implied, immediate, absolute, absolute-X, absolute-Y, variable, direct label, indirect, and indirect variable.

IMPLIED ADDRESSING

```
LABEL:TXA -0000 (COMMENT)
```

Implied addressing is used when the operator does not need an operand. In the line above, transfer X-register to accumulator does not need any operand. A dash is used as the addressing technique, and any four-digit number is used as the operand.

IMMEDIATE ADDRESSING

```
LABEL: LDA #0521 (COMMENT)
```

Immediate addressing is used when the data to be used is in the instruction itself. In the line above, the

accumulator is loaded with the value 521. A "#" is used as the addressing technique, and any four digit number is used as the operand.

ABSOLUTE ADDRESSING

LABEL: LDA $0065 (Comment)

In absolute addressing, the system takes the value of the stack location specified by the operand. In the line above, the accumulator is loaded with the contents of stack location 65. A dollar sign is used as the addressing technique, and any four digit number is used as the operand, as long as the number does not exceed the stack limit.

ABSOLUTE-X and -Y ADDRESSING

LABEL: LDA X0065 (Comment)

In absolute-X addressing, the value of the operand is added to the value in the X-register, and that value in turn is used as the value of the stack location to be used. In the above line, the accumulator is loaded with the contents of stack location (65 + X), where X is the value stored in the X-register. Absolute-Y addressing uses the same procedure, except the Y-register is used rather than the X-register. An X and Y is used for the X-and Y-addressing technique, respectively. The operand is any four-digit number.

VARIABLE ADDRESSING

LABEL: LDA V VAR1 (Comment)

Variable addressing is similar to absolute addressing, the one change being that a variable, rather than a stack address, is used as the operand. A "V" is used as the addressing technique. A variable name is used as the operand.

DIRECT ADDRESSING

LABEL: BNE $0015 (Comment)

In direct addressing, the operand signifies a specific line number to be loaded into the program counter. In the above example, if the accumulator is not equal to zero, the program counter is loaded with the value 15, and the program will jump to line 15. (This is not the EDITOR generated number, but the 15th significant line of the program.) Like absolute addressing, a dollar sign is used as the addressing technique, and any four-digit number can be used as the operand.

LABELED ADDRESSING

LABEL: BNE L LAB1 (Comment)

Labeled addressing is the same as direct addressing, except that the label of a line is used instead of a line number. In the above line, if the accumulator is not equal to zero, the program counter is loaded with the line number

labeled LAB1. An "L" is used as the addressing technique, and a label name is used as the operand.

INDIRECT ADDRESSING

LABEL: LDA I 0034 (Comment)

Indirect addressing uses one memory location as a pointer to another memory location. The second memory location is then used as the operand. In the example, the accumulator is loaded with the value stored in stack location D(34), where D(x) is the contents of stack location x. Therefore, if stack location 34 contains a value of, say 48, then the system would load the contents of stack location 48 into the accumulator. An "I" is used as the addressing technique, and the operand is any four digit number which does not go beyond the stack limit.

INDIRECT VARIABLE ADDRESSING

LABEL: LDA P VAR1 (Comment)

Indirect variable addressing is the same as indirect addressing, except that a variable location is used, rather than an literal stack position. In the example above, VAR1 is used as a pointer to another position on the stack, say 48. The contents of stack location 48 would then be loaded into the accumulator. A "P" is used as the addressing technique, and the operand can be any variable name.

EXAMPLES OF SYNTAX

The following are correct statements:

```
LDA VVARIABLES1
STA $   0002 (Comment)
      END:BRK -0000
1234:    INP $0000
```

The following are incorrect:

```
LDA$0001            (No space after operator)
LABEL5 :ADC $0001   (Space between label and colon)
BRK                 (No addressing technique or operand)
START:ADC V 0001    (Variable name required as operand)
BNE $0001(START)    (No space between operand and comment)
```

PART III: EXECUTION AND OPERATORS

OPERATORS

The execution of any program is principally dependent on the operators. Once the simulator retreives the proper operand, the operator determines what is done with it. The following is a list of the operators available to the user:

ADD ACCUMULATOR (ADC) ADDRESSING: #,$,X,Y,V,I,P

This function adds the value of the operand to the accumulator. If the result is greater than 256, the accumulator "rolls over", and the overflow flag is changed.

AND ACCUMULATOR (AND) ADDRESSING: #,$,X,Y,V,I,P

This function takes the binary value of both the operand and the accumulator, and performs a logical "AND" operation on each corresponding bit. The result is stored into the accumulator.

ARITHMETIC SHIFT LEFT (ASL) ADDRESSING:$,X,Y,V,I,P

This function takes the binary value of the memory location specified by the operand, and shifts each bit to the left one position. The leftmost bit is shifted out, and a zero is shifted in to the right.

BRANCH EQUAL TO ZERO (BEQ) ADDRESSING:#,L

This function loads the program counter with the value specified by the operand, but only if the zero flag is set (Z=1). If the zero flag is not set, then the program continues to the next line.

BRANCH RESULT MINUS (BMI) ADDRESSING: #,L

This function loads the program counter with the value specified by the operand, but only if the negative flag is set (N=1). Otherwise, the program continues to the next line.

BRANCH NOT EQUAL TO ZERO (BNE) ADDRESSING: #,L

This function loads the program counter with the value specified by the operand, but only if the zero flag is not set (Z=0). Otherwise, the program continues with the next line.

BRANCH RESULT PLUS (BPL) ADDRESSING:#,L

This function loads the program counter with the value specified by the operand, but only if the positive flag is set (P=1). Otherwise, the program continues with the next line.

BREAK (BRK) ADDRESSING: -

This function stops the execution of the program and returns the user to the MPE operating system. From there,

the user can abort the program, or use the :RESUME command to continue.

BRANCH OVERFLOW CLEAR (BVC) ADDRESSING: #,L

This function loads the program counter with the value specified by the operand, but only if the overflow flag is clear (V=0). Otherwise, the program continues with the next line.

BRANCH OVERFLOW SET (BVS) ADDRESSING: #,L

This function loads the program counter with the value specified by the operand, but only if the overflow flag is set (V=1). Otherwise, the program continues with the next line.

CARRIAGE CONTROL SET (CCT) ADDRESSING: #,$,X,Y,V,I,P

This function changes the carriage control for the PRT and PRA commands. The value in the operand specifies the number of carriage returns to be executed after any subsequent output statements as follows:

0-No carriage return/or line feed

1-Carriage return and one line feed

2-Carriage return and two line feeds

3-Carriage return and skip to the next page

The default value is 1.

CLEAR OVERFLOW FLAG (CLV) ADDRESSING: -

This function sets the overflow flag to zero (V=0).

DECREMENT MEMORY (DEC) ADDRESSING: $,X,Y,V,I,P

This function subtracts 1 from the stack location specified by the operand.

DECREMENT X-REGISTER (DEX) ADDRESSING: -

This function subtracts 1 from the X-register.

DECREMENT Y-REGISTER (DEY) ADDRESSING: -

This function subtracts 1 from the Y-register.

EXECUTIVE OR ACCUMULATOR (EOR) ADDRESSING: #,$,X,Y,V,I,P

This function takes the binary value of both the accumulator and the operand, and performs a logical exclusive "OR" operation on each corresponding bit. The result is stored into the accumulator.

INPUT ASCII CHARACTER (INA) ADDRESSING: $,X,Y,V,I,P

This function accepts a value from the standard input device (usually a terminal) and stores its ASCII code into the stack location specified by the operand.

INCREMENT MEMORY (INC) ADDRESSING: $,X,Y,V,I,P

This function adds 1 to the stack location specified by the operand.

INPUT NUMERICAL VALUE (INP) ADDRESSING: $,X,Y,V,I,P

This function accepts a numerical value from the standard input device (usually a terminal), and stores it into the stack location specified by the operand.

INCREMENT X-REGISTER (INX) ADDRESSING: -

This function adds 1 to the X-register.

INCREMENT Y-REGISTER (INY) ADDRESSING: -

This function adds 1 to the Y-register.

JUMP (JMP) ADDRESSING: #,L

This function loads the program counter with the value specified by the operand.

JUMP SAVING RETURN ADDRESS (JSR) ADDRESSING: #,L

This function "pushes" the value of the program counter onto the stack, then loads the program counter with the value specified by the operand. This allows the use of subroutines.

LABEL DECLARATION (LAB) ADDRESSING: L

This function declares the operand to be a label. This operator is not executable, but is simply a declaration which must be made before the label is referred to in any way.

LOAD ACCUMULATOR (LDA) ADDRESSING: #,$,X,Y,V,I,P

This function takes the value specified by the operand and stores it into the accumulator.

LOAD X-REGISTER (LDX) ADDRESSING: #,$,V,I,P

This function takes the value specified by the operand and stores it into the X-register.

LOAD Y-REGISTER (LDY) ADDRESSING: #,$,V,I,P

This function takes the value specified by the operand and stores it into the Y-register.

LOGICAL SHIFT RIGHT (LSR) ADDRESSING: $,X,Y,V,I,P

This function takes the binary value of the stack location specified by the operand, and shifts each bit by 1 to the right. The rightmost bit is shifted out, and a zero is shifted in to the left.

NO OPERATION (NOP) ADDRESSING: -

This function specifies that no operation is to be executed. The program simply goes on to the next line.

OR ACCUMULATOR (ORA) ADDRESSING: #,$,X,Y,V,I,P

This function takes the binary values of both the accumulator and the value specified by the operand, and performs a logical "OR" operation. The result is stored into the accumulator.

PUSH ACCUMULATOR ONTO STACK (PHA) ADDRESSING: -

This function "pushes" the accumulator onto the top of the stack, and decrements the top-of-stack pointer.

PUSH PROGRAM STATUS ONTO STACK (PHP) ADDRESSING: -

This function "pushes" the status register onto the top of the stack, and decrements the top-of-stack pointer.

PULL ACCUMULATOR FROM STACK (PLA) ADDRESSING: -

This function "pops" a value from the top of the stack, stores it into the accumulator, and increments the top-of-stack pointer.

PULL PROGRAM STATUS FROM STACK (PLP) ADDRESSING: -

This function "pops" a value from the top of the stack, stores it into the status register, and increments the top-of-stack pointer.

PRINT ASCII CHARACTER (PRA) ADDRESSING: $,X,Y,V,I,P

This function takes the value specified by the operand, and prints its ASCII equivalent onto the standard listing device.

PRINT NUMERICAL VALUE (PRT) ADDRESSING: $,X,Y,V,I,P

This function takes the value specified by the operand, and prints it onto the standard listing device.

RETURN FROM SUBROUTINE (RTS) ADDRESSING: -

This function takes the value from the top of the stack (previously pushed by a JSR command), and stores it into the program counter. This allows the user to return from a subroutine.

SUBTRACT FROM ACCUMULATOR (SBC) ADDRESSING: #,$,X,Y,V,I,P

This function takes the value specified by the operand and subtracts it from the accumulator. If the result is less than -255, then the accumulator "rolls over" and the overflow flag is changed.

STORE INTO ACCUMULATOR (STA) ADDRESSING: #,$,X,Y,V,I,P

This function takes the value specified by the operand and stores it into the accumulator.

STOP (STP) ADDRESSING: -

This function stops execution of a program and returns the user to the MPE operation system.

STORE INTO X-REGISTER (STX) ADDRESSING: #,$,V,I,P

This function takes the value specified by the operand, and stores it into the X-register.

STORE INTO Y-REGISTER (STY) ADDRESSING: #,$,V,I,P

This function takes the value specified by the operand, and stores it into the Y-register.

TRANSFER ACCUMULATOR TO X-REGISTER (TAX) ADDRESSING: -

This function transfers the contents of the accumulator into the X-register. The contents of the accumulator are unaffected.

TRANSFER ACCUMULATOR TO Y-REGISTER (TAY) ADDRESSING: -

This function transfers the contents of the accumulator into the Y-register. The contents of the accumulator are unaffected.

TRANSFER STATUS REGISTER TO X-REGISTER (TSX) ADDRESSING: -

This function transfers the contents of the program status register to the X-register. The status register remains the same.

TRANSFER X-REGISTER TO ACCUMULATOR (TXA) ADDRESSING: -

This function transfers the contents of the X-register to the accumulator. The X-register remains the same.

TRANSFER X-REGISTER TO STATUS REGISTER (TXS) ADDRESSING: -

This function transfers the contents of the X-register to the status register. The contents of the X-register remains the same.

TRANSFER Y-REGISTER TO ACCUMULATOR (TYA) ADDRESSING: -

This function transfers the contents of the Y-register to the accumulator. The contents of the Y-register are unaffected.

VARIABLE DECLARATION (VAR) ADDRESSING: V

This function declares the operand to be a variable. This operand is not executable, but is simply a declaration which must be made before the variable is referred to in any way.

PART IV: EXECUTION OF THE SIMULATOR

The simulator is stored in object form on the disk. It takes an EDITOR-created file of source code, parses the code, then puts it into a form that it can execute quickly. This code is stored in a job-temporary file, so the contents of the original source file remains unchanged. If the simulator has detected no syntax arrors, then it uses this temporary "object" file to execute the program. After inputting the program onto an EDITOR file, the user activates the simulator from the main operating system by using the following command:

:ASSEMBLEGO progname

Where progname is the EDITOR-file which contains the source code. Remember, there should be no blank lines in this file, and no line should exceed 72 characters. The simulator will print out the source listing, including the line numbers. These line numbers are not the EDITOR line numbers, but an ordinal value signifying the nth significant line in the program. LAB and VAR commands are not considered significant.

After the source listing the simulator prints the number of errors and warnings detected. If no errors are detected, then the simulator prepares the program for execution. If no errors are yet detected, the program executes.

Appendix C shows a list of all error and warning messages. Appendix D shows a sample run.

DISCLAIMER

This simulator and manual fulfills, in part, the requirements for a three-semester honors course. In preparing this project, I had to always weigh the relative advantages to the amount of time it would take to accomplish a certain goal. I did not have the time to implement everything that I wished to see in an assembly language; however, I believe I have created a useful and functional system which achieves what I had hoped to accomplish.

One aspect of this project which I was not able to accomplish to my satisfaction was the debugging procedure. It is impossible to test this simulator under every possible situation. No doubt the user will find errors. I will make every reasonable effort to correct those errors up through May, 1983. After that, it may be a good idea to devote another honors project or applied programming course to the upkeep and expansion of this system.

BIBLIOGRAPHY

Abrams, Marshall and Stein, Phillip. Computer Hardware and Software: An Interdisciplinary Introduction. Reading, Mass.: Wesley Publishing Company, 1973.

Camp, R. C., Smay, T. A., and Triska, C. J. Microprocessor System Engineering. Beaverton, Oregon: Matrix Publishers, 1979.

Machine Introduction Set Reference Manual. Santa Clara, California: Hewlett-Packard Company, 1976.

MPE Intrinsics Reference Manual. Santa Clara, California: Hewlett-Packard Company, 1978.

Systems Programming Language Reference Manual. Santa Clara California: Hewlett-Packard Company, 1976.

Systems Programming Language Textbook. Santa Clara, California: Hewlett-Packard Company, 1976.

APPENDIX A: ADDRESSING TECHNIQUES

SYMBOL	TECHNIQUE	OPERAND TYPE
-	Implied	Numerical
#	Immediate, direct	Numerical
$	Absolute	Numerical
X	Absolute-X	Numerical
Y	Absolute-Y	Numerical
V	Variable	Variable
L	Labeled	Label
I	Indirect	Numerical
P	Variable Indirect	Variable

APPENDIX B: OPERATORS

Operator	Function	Addressing Technique									System Code
		-	#	$	X	Y	V	L	I	P	
ADC	A + M -) A		x	x	x	x	x		x	x	70
AND	A AND M -) A		x	x	x	x	x		x	x	71
ASL	O (- M (- 0			x	x	x	x		x	x	60
BEQ	BRANCH Z = 1		x					x			32
BMI	BRANCH N = 1		x					x			33
BNE	BRANCH Z = 0		x					x			34
BPL	BRANCH P = 1		x					x			35
BRK	BREAK	x									1
BVC	BRANCH V = 0		x					x			36
BVS	BRANCH V = 1		x					x			37
CCT	CARRIAGE CONTROL		x	x	x	x	x	x	x	x	76
CLV	O -) V	x									3
DEC	M - 1 -) M			x	x	x	x		x	x	63
DEX	X - 1 -) X	x									4
DEY	Y - 1 -) Y	x									5
EOR	A EOR M -) A		x	x	x	x	x		x	x	72
INA	$STDIN -) M			x	x	x	x		x	x	68
INC	M + 1 -) M			x	x	x	x		x	x	61
INP	$STDIN -) M			x	x	x	x		x	x	66
INX	X + 1 -) X	x									6
INY	Y + 1 -) Y	x									7
JMP	M -) PC		x					x			40
JSR	M -) PC -) ST		x					x			41
LAB	DECLARATION		x	x	x	x	x		x	x	80

Operator	Function	Addressing Technique									System Code
		-	#	$	X	Y	V	L	I	P	
LDA	M -) A		x	x			x		x	x	73
LDX	M -) X		x	x			x		x	x	50
LDY	M -) Y		x	x			x		x	x	51
LSR	O -) M -) 0			x	x	x	x		x	x	62
NOP	NO OPERATION	x									8
ORA	M OR A -) A		x	x	x	x	x		x	x	74
PHA	A -) ST	x									9
PHP	PC -) ST	x									10
PLA	ST -) A	x									11
PLP	ST -) PC	x									12
PRA	M -) $STDLIST			x	x	x	x		x	x	67
PRT	M -) $STDLIST			X	X	X	X		X	X	65
RTS	ST -) PC	x									13
SBC	A - M -) A		x	x	x	x	x		x	x	75
STA	A -) M		x	x	x	x	x		x	x	64
STP	STOP	x									21
STX	X -) M		x	x			x		x	x	52
STY	Y -) M		x	x			x		x	x	53
TAX	A -) X	x									15
TAY	A-) Y	x									16
TSX	S -) X	x									17
TXA	X-) A	x									18
TXS	X -) S	x									19
TYA	Y -) A	x									20
VAR	DECLARATION						X				81

APPENDIX C: ERROR MESSAGES AND RECOVERY

Error No.	Message
0	System simulator error - see instructor
1	Operator less than 3 Characters - check spacing
2	Operand less than 4 Characters - check spacing
3	Unrecognized operator - check Appendix B
4	Undefined variable name - VAR command needed
5	Undefined label name - LAB command needed
6	Unrecognizable Data - ???
7	End-of-line detected after label read
8	End-of-line detected after operator read
9	End-of-line detected before operand read
10	Implied (-) addressing only
11	Labeled (L) or direct (#) addressing only
12	Labeled (L) or direct (#) addressing only
13	Improper addressing technique - see Appendix A
14	Duplicate label name - label already defined
15	Duplicate variable name - variable already defined
16	Missing label - branch to label not associated with a line
17	Branch to non-existent statement
18	Improper operand - see addressing techniques

Warning No.	Message
0	System simulator error - see instructor
1	Operator greater than 3 Characters - check spacing
2	Operand greater than 4 Characters - check spacing
3	Carriage control number greater than 3 -single spacing assumed

APPENDIX D: SOURCE LISTING

```
PAGE 1      HEWLETT-PACKARD 32201A.7.12 EDIT/3000 WED, MAR 30, 1983,  5:30 PM (C) HE

    1       $CONTROL USLINIT,NOWARN
    2       BEGIN
    3       INTRINSIC FOPEN,FCLOSE,PRINT,FREAD,FWRITEDIR,READ,BINARY,ASCII,DATELI"";
    4       INTRINSIC FREADDIR;
    5       INTEGER LINENUMBER,     << PROGRAM COUNTER >>
    6               LABELNUMBER,    << FIRST EMPTY POSITION IN LABEL LIST >>
    7               VARIABLENUMBER, << FIRST EMPTY POSITION IN LABEL LIST >>
    8               ERROR,          << CURRENT ERROR NUMBER >>
    9               ERRORS,         << TOTAL NUMBER OF ERRORS >>
   10               WARNING,        << CURRENT WARNING NUMBER >>
   11               WARNINGS,       << TOTAL NUMBER OF WARNINGS >>
   12               SOURCENUMBER,   << SOURCEFILE NUMBER >>
   13               OBJECTNUMBER,   << OBJECTFILE NUMBER >>
   14               LOOP,           << PROGRAM LOOPS >>
   15               LOOP1,
   16               STACKPOINTER,   << PROGRAM STACK POINTER >>
   17               XREG,           << X-REGISTER >>
   18               YREG,           << Y-REGISTER >>
   19               STATUS,         << STATUS REGISTER >>
   20               COUNTER,        << CURRENT LINE NUMBER DURING EXECUTION >>
   21               OPERAND,        << OPERAND DURING EXECUTION >>
   22               CURSOR,         << POINTER USED TO PARSE LINE >>
   23               LENGTH,         << LENGTH OF I-O STRINGS >>
   24               ADDRESSING;     << ADDRESSING OBJECT CODE >>
   25       INTEGER  OPLIMIT:=100;   << MAXIMUM STACK SIZE >>
   26       DOUBLE   DOUBLENUMBER;   << DOUBLE INTEGER BUFFER FOR FWRITEDIR >>
   27       INTEGER  ARRAY
   28               STACK(0:101),  << PROGRAM STACK >>
   29               CODE(0:2), << 'OBJECT' CODE >>
   30               LABELREF(0:19);<<LABEL REFERENCE STACK>>
   31       BYTE ARRAY
   32               SOURCEFILE(0:7):="SOURCE  ";    << SOURCEFILE NAME >>
   33       BYTE ARRAY
   34               OBJECTFILE(0:7):="OBJECT  ";    << OBJECTFILE NAME >>
   35       BYTE ARRAY
   36               LINE(0:72),     << SOURCELINE READ IN >>
   37               LABELS(0:199), << LABEL LIST >>
   38               VARIABLES(0:199),<< VARIABLE LIST >>
   39               OPERATOR(0:3), << PARSED OPERATOR >>
   40               OPBYTE(0:3),   << PARSED OPERAND IN DECIMAL FORM >>
   41               OPLABEL(0:9),  << PARSED OPERAND IN LABEL FORMAT >>
   42               OPVARIABLE(0:9),<<PARSED OPERAND IN VARIABLE FORMAT >>
   43               TECHNIQUE(0:0),<< PARSED TECHNIQUE CODE >>
   44               DISPLAYLINE(0:79);<< DISPLAY LINE >>
   45       LOGICAL  EOF:= FALSE;    << END OF FILE TEST >>
   46       LOGICAL  TESTVARIABLE,   <<TEST AREA FOR VARIABLES>>
   47                TESTLABEL;      << TEST AREA FOR LABELS>>
   48
   49       INTEGER PROCEDURE OPCODE(OPERATOR);
   50       << THIS PROCEDURE TAKES THE PARSED OPCODE AND ASSIGNS AN
   51          INTEGER WHICH IS USED BY THE PROGRAM TO DETERMINE THE
   52          PROPER ACTION (USING A CASE STATEMENT). IF THE OPCODE
   53          IS UNRECOGNIZABLE, THE PROCEDURE RETURNS A ZERO.   >>
   54
   55       BYTE ARRAY OPERATOR;
   56       BEGIN
   57           INTEGER CODE;
```

```
    58          CODE:=0;
    59          OPERATOR(3):=" ";
    60        <<OP CODES THAT CAN ONLY USE IMPLIED ADDRESSING>>
    61          IF OPERATOR = "BRK " THEN CODE:=1;
    62          IF OPERATOR = "CLV " THEN CODE:=3;
    63          IF OPERATOR = "DEX " THEN CODE:=4;
    64          IF OPERATOR = "DEY " THEN CODE:=5;
    65          IF OPERATOR = "INX " THEN CODE:=6;
    66          IF OPERATOR = "INY " THEN CODE:=7;
    67          IF OPERATOR = "NOP " THEN CODE:=8;
    68          IF OPERATOR = "PHA " THEN CODE:=9;
    69          IF OPERATOR = "PHP " THEN CODE:=10;
    70          IF OPERATOR = "PLA " THEN CODE:=11;
    71          IF OPERATOR = "PLP " THEN CODE:=12;
    72          IF OPERATOR = "RTS " THEN CODE:=13;
    73          IF OPERATOR = "TAX " THEN CODE:=15;
    74          IF OPERATOR = "TAY " THEN CODE:=16;
    75          IF OPERATOR = "TSX " THEN CODE:=17;
    76          IF OPERATOR = "TXA " THEN CODE:=18;
    77          IF OPERATOR = "TXS " THEN CODE:=19;
    78          IF OPERATOR = "TYA " THEN CODE:=20;
    79          IF OPERATOR = "STP " THEN CODE:=21;
    80        <<OP CODES THAT CAN ONLY USE LABEL OR RELATIVE ADDRESSING>>
    81          IF OPERATOR = "BEQ " THEN CODE:=32;
    82          IF OPERATOR = "BMI " THEN CODE:=33;
    83          IF OPERATOR = "BNE " THEN CODE:=34;
    84          IF OPERATOR = "BPL " THEN CODE:=35;
    85          IF OPERATOR = "BVC " THEN CODE:=36;
    86          IF OPERATOR = "BVS " THEN CODE:=37;
    87        <<OP CODES THAT CAN ONLY USE LABEL OR ABSOLUTE ADDRESSING>>
    88          IF OPERATOR = "JMP " THEN CODE:=40;
    89          IF OPERATOR = "JSR " THEN CODE:=41;
    90        << CAN USE IMMEDIATE, ABSOLUTE, VARIABLE, OR INDIRECT ADDRESSING>>
    91          IF OPERATOR = "LDX " THEN CODE:=50;
    92          IF OPERATOR = "LDY " THEN CODE:=51;
    93          IF OPERATOR = "STX " THEN CODE:=52;
    94          IF OPERATOR = "STY " THEN CODE:=53;
    95        <<CAN USE ABSOLUTE, ABS X OR Y, VARIABLE, OR INDIRECT>>
    96          IF OPERATOR = "ASL " THEN CODE:=60;
    97          IF OPERATOR = "INC " THEN CODE:=61;
    98          IF OPERATOR = "LSR " THEN CODE:=62;
    99          IF OPERATOR = "DEC " THEN CODE:=63;
   100          IF OPERATOR = "STA " THEN CODE:=64;
   101          IF OPERATOR = "PRT " THEN CODE:=65;
   102          IF OPERATOR = "INP " THEN CODE:=66;
   103          IF OPERATOR = "PRA " THEN CODE:=67;
   104          IF OPERATOR = "INA " THEN CODE:=68;
   105        <<IMMEDIATE,ABSOLUTE,ABS X OR Y, VARIABLE, OR INDIRECT>>
   106          IF OPERATOR = "ADC " THEN CODE:=70;
   107          IF OPERATOR = "AND " THEN CODE:=71;
   108          IF OPERATOR = "EOR " THEN CODE:=72;
   109          IF OPERATOR = "LDA " THEN CODE:=73;
   110          IF OPERATOR = "ORA " THEN CODE:=74;
   111          IF OPERATOR = "SBC " THEN CODE:=75;
   112          IF OPERATOR = "CCT " THEN CODE:=76;
   113        <<LABEL AND VARIABLE DECLARATIONS ONLY>>
   114          IF OPERATOR = "LAB " THEN CODE:=80;
```

```
115         IF OPERATOR = "VAR " THEN CODE:=81;
116       OPCODE:=CODE
117     END;
118
119     PROCEDURE PARSER(LINENUMBER,LINE,LABELS,LABELNUMBER,OPERATOR,
120         TECHNIQUE,OPBYTE,OPLABEL,OPVARIABLE,VARIABLES,ERROR,WARNING);
121     INTEGER LINENUMBER,LABELNUMBER,ERROR,WARNING;
122     BYTE ARRAY LINE,LABELS,OPERATOR,TECHNIQUE,OPBYTE;
123     BYTE ARRAY OPLABEL,OPVARIABLE,VARIABLES;
124     BEGIN
125     INTEGER LAST;
126     INTEGER LOOP;
127         ERROR:=0;
128         WARNING:=0;
129         LAST:=0;
130         MOVE OPERATOR:=4(" ");
131         TECHNIQUE(0):=" ";
132         WHILE (LINE(LAST) = " ") AND (LAST < 72) DO LAST:=LAST+1;
133         IF LAST = 72 THEN
134         BEGIN
135             ERROR:=6;
136             RETURN
137         END;
138         CURSOR:=LAST;
139         WHILE (LINE(CURSOR)<>" ") AND (LINE(CURSOR)<>":") AND
140             (CURSOR<72) DO CURSOR:=CURSOR+1;
141         IF CURSOR =  72 THEN
142         BEGIN
143             ERROR:=6;
144             RETURN
145         END;
146     << EXTRACT LABEL IF IT EXISTS >>
147         IF LINE(CURSOR) = ":" THEN
148         BEGIN
149             IF CURSOR-LAST > 10 THEN
150                 FOR LOOP:=0 UNTIL 9 DO
151                     OPLABEL(LOOP):=LINE(LAST+LOOP)
152             ELSE BEGIN
153                 FOR LOOP:=0 UNTIL 9 DO
154                     OPLABEL(LOOP):=" ";
155                 FOR LOOP:=0 UNTIL (CURSOR-LAST)-1 DO
156                     OPLABEL(LOOP):=LINE(LAST+LOOP)
157             END;
158             FOR LOOP:=0 STEP 10 UNTIL LABELNUMBER DO
159             BEGIN
160                 TESTLABEL:= TRUE;
161                 FOR LOOP1:=0 UNTIL 9 DO
162                     IF LABELS(LOOP+LOOP1)<>OPLABEL(LOOP1) THEN
163                     TESTLABEL:= FALSE;
164                 IF TESTLABEL THEN
165                     LABELREF(LOOP/10):=LINENUMBER
166             END;
167             CURSOR:=CURSOR+1;
168             LAST:=CURSOR;
169             WHILE (LINE(LAST)=" ") AND (LAST<72) DO LAST:=LAST+1;
170             IF LAST = 72 THEN
171             BEGIN
```

```
172                 ERROR:=7;
173                 RETURN
174             END;
175             CURSOR:=LAST;
176             WHILE (LINE(CURSOR)<>" ") AND (CURSOR<72) DO
177                 CURSOR:=CURSOR+1;
178             IF CURSOR = 72 THEN
179             BEGIN
180                 ERROR:=7;
181                 RETURN
182             END;
183         END;
184     << EXTRACT OPERATOR >>
185         IF CURSOR-LAST > 3 THEN WARNING:=1;
186         IF CURSOR-LAST < 3 THEN
187         BEGIN
188             ERROR:=1;
189             RETURN
190         END;
191         FOR LOOP:=0 UNTIL 2 DO
192             OPERATOR(LOOP):=LINE(LAST+LOOP);
193         CURSOR:=CURSOR+1;
194         LAST:=CURSOR;
195         WHILE (LINE(LAST)=" ") AND (LAST<72) DO LAST:=LAST+1;
196         IF LAST = 72 THEN
197         BEGIN
198             ERROR:=8;
199             RETURN
200         END;
201     << EXTRACT ADDRESSING TECHNIQUE>>
202         TECHNIQUE:=LINE(LAST);
203         LAST:=LAST+1;
204         WHILE (LINE(LAST)=" ") AND (LAST<72) DO LAST:=LAST+1;
205         IF LAST =72 THEN
206         BEGIN
207             ERROR:=9;
208             RETURN
209         END;
210         CURSOR:=LAST;
211         WHILE (LINE(CURSOR)<>" ") AND (CURSOR<72)  DO
212             CURSOR:=CURSOR+1;
213         IF CURSOR = 72 THEN
214         BEGIN
215             ERROR:=9;
216             RETURN
217         END;
218     << FIND APPROPRIATE OPERAND FOR ADDRESSING TECHNIQUE >>
219         IF (TECHNIQUE = "#") OR (TECHNIQUE = "$") OR (TECHNIQUE = "X")
220           OR (TECHNIQUE = "Y") OR (TECHNIQUE = "I") THEN
221         BEGIN
222             IF CURSOR-LAST > 4 THEN WARNING:=2;
223             IF CURSOR-LAST < 4 THEN
224             BEGIN
225                 ERROR:=2;
226                 RETURN
227             END;
228             FOR LOOP:=0 UNTIL 3 DO
```

```
229                 OPBYTE(LOOP):=LINE(LAST+LOOP)
230             END;
231             IF (TECHNIQUE = "V") OR (TECHNIQUE = "D") THEN
232                 IF CURSOR-LAST > 10 THEN
233                     FOR LOOP:=0 UNTIL 9 DO
234                         OPVARIABLE(LOOP):=LINE(LAST+LOOP)
235                 ELSE BEGIN
236                     FOR LOOP:=0 UNTIL 9 DO
237                         OPVARIABLE(LOOP):=" ";
238                     FOR LOOP:=0 UNTIL CURSOR-LAST DO
239                         OPVARIABLE(LOOP):=LINE(LAST+LOOP)
240                 END;
241             IF TECHNIQUE = "L" THEN
242                 IF CURSOR-LAST > 10 THEN
243                     FOR LOOP:=0 UNTIL 9 DO
244                         OPLABEL(LOOP):=LINE(LAST+LOOP)
245                 ELSE BEGIN
246                     FOR LOOP:=0 UNTIL 9 DO
247                         OPLABEL(LOOP):=" ";
248                     FOR LOOP:=0 UNTIL CURSOR-LAST DO
249                         OPLABEL(LOOP):=LINE(LAST+LOOP)
250                 END;
251         END;
252
253         PROCEDURE PRINTERROR(ERROR,ERRORS);
254             INTEGER ERROR,ERRORS;
255         BEGIN
256             BYTE ARRAY ERRORLINE(0:71);
257             MOVE ERRORLINE:=72(" ");
258             CASE ERROR OF BEGIN
259                 MOVE ERRORLINE:="SYSTEM SIMULATOR ERROR";
260                 MOVE ERRORLINE:="ERROR 1: OPERATOR < 3 CHARACTERS";
261                 MOVE ERRORLINE:="ERROR 2: OPERAND < 4 CHARACTERS";
262                 MOVE ERRORLINE:="ERROR 3: UNRECOGNIZED OPERATOR";
263                 MOVE ERRORLINE:="ERROR 4: UNDEFINED VARIABLE NAME";
264                 MOVE ERRORLINE:="ERROR 5: UNDEFINED LABEL NAME";
265                 MOVE ERRORLINE:="ERROR 6: UNRECOGNIZABLE DATA";
266                 MOVE ERRORLINE:="ERROR 7: EOLN DETECTED AFTER LABEL READ";
267                 MOVE ERRORLINE:="ERROR 8: EOLN DETECTED AFTER OPERATOR READ";
268                 MOVE ERRORLINE:="ERROR 9: EOLN DETECTED BEFORE OPERAND READ";
269                 MOVE ERRORLINE:="ERROR 10: IMPLIED ADDRESS ONLY";
270                 MOVE ERRORLINE:="ERROR 11: LABEL OR IMMEDIATE ADDRESS ONLY";
271                 MOVE ERRORLINE:="ERROR 12: LABEL OR IMMEDIATE ADDRESS ONLY";
272                 MOVE ERRORLINE:="ERROR 13: IMPROPER ADDRESSING TECHNIQUE";
273                 MOVE ERRORLINE:="ERROR 14: DUPLICATE LABEL NAME";
274                 MOVE ERRORLINE:="ERROR 15: DUPLICATE VARIABLE NAME";
275                 MOVE ERRORLINE:="ERROR 16: MISSING LABEL";
276                 MOVE ERRORLINE:="ERROR 17: BRANCH TO NON-EXISTENT STATEMENT";
277                 MOVE ERRORLINE:="ERROR 18: IMPROPER OPERAND";
278             END;
279         PRINT(ERRORLINE,-72,"0");
280         ERRORS:=ERRORS+1;
281         ERROR:=0;
282         END;
283
284         PROCEDURE PRINTWARNING(WARNING,WARNINGS);
285             INTEGER WARNING,WARNINGS;
```

```
286       BEGIN
287           BYTE ARRAY WARNINGLINE(0:71);
288           MOVE WARNINGLINE:=72(" ");
289           CASE WARNING OF BEGIN
290                MOVE WARNINGLINE:="SYSTEM SIMULATOR ERROR";
291                MOVE WARNINGLINE:="WARNING 1: OPERATOR >3 CHARACTERS";
292                MOVE WARNINGLINE:="WARNING 2: OPERAND > 4 CHARACTERS";
293                MOVE WARNINGLINE:="WARNING 3: CARRIAGE CONTROL CODE NUMBER > 64";
294                MOVE WARNINGLINE:="WARNING 4";
295           END;
296       PRINT(WARNINGLINE,-72,"0");
297       WARNINGS:=WARNINGS+1;
298       WARNING:=0;
299       END;
300
301       PROCEDURE MESSAGE;
302       << PRINTS SYSTEM ERROR MESSAGES FOR ANY I/O ERRORS >>
303       BEGIN
304           INTEGER ERRORCODE,DUMMY;
305           BYTE ARRAY ERRORLINE(0:71);
306           INTEGER NUMBER:=0;
307           INTRINSIC FCHECK, FERRMSG;
308           FCHECK(NUMBER,ERRORCODE);
309           MOVE ERRORLINE:=72(" ");
310           FERRMSG(ERRORCODE,ERRORLINE,DUMMY);
311           PRINT(ERRORLINE,-72," ")
312       END;
313
314       PROCEDURE ADDLABEL(LABELS,LABELNUMBER,OPLABEL);
315       << ADDS A LABEL TO THE LABEL TABLE AND CHECKS FOR DUPLICATION >>
316       BYTE ARRAY LABELS,OPLABEL;
317       INTEGER LABELNUMBER;
318       BEGIN
319           FOR LOOP:=0 STEP 10 UNTIL LABELNUMBER DO
320           BEGIN
321                TESTLABEL:= TRUE;
322                FOR LOOP1:=0 UNTIL 9 DO
323                     IF LABELS(LOOP+LOOP1) <> OPLABEL(LOOP1) THEN
324                          TESTLABEL:= FALSE;
325                IF TESTLABEL THEN
326                BEGIN
327                     ERROR:=14;
328                     PRINTERROR(ERROR,ERRORS);
329                     RETURN
330                END;
331           END;
332           FOR LOOP:=0 UNTIL 9 DO
333                LABELS(LABELNUMBER+LOOP):=OPLABEL(LOOP);
334           LABELNUMBER:=LABELNUMBER + 10
335       END;
336
337       PROCEDURE ADDVARIABLE(VARIABLES,VARIABLENUMBER,OPVARIABLE);
338       << ADDS A VARIABLE TO THE VARIABLE TABLE AND ASSIGNS A
339          POSITION IN THE STACK >>
340       BYTE ARRAY VARIABLES,OPVARIABLE;
341       INTEGER VARIABLENUMBER;
342       BEGIN
```

```
343          FOR LOOP:=0 STEP 10 UNTIL VARIABLENUMBER DO
344          BEGIN
345              TESTVARIABLE:= TRUE;
346              FOR LOOP1:=0 UNTIL 9 DO
347                  IF VARIABLES(LOOP+LOOP1) <> OPVARIABLE(LOOP1) THEN
348                      TESTVARIABLE:= FALSE;
349              IF TESTVARIABLE THEN
350              BEGIN
351                  ERROR:=15;
352                  PRINTERROR(ERROR,ERRORS);
353                  RETURN
354              END;
355          END;
356          FOR LOOP:=0 UNTIL 9 DO
357              VARIABLES(VARIABLENUMBER+LOOP):=OPVARIABLE(LOOP);
358          VARIABLENUMBER:=VARIABLENUMBER + 10
359      END;
360
361      PROCEDURE PREPARE(LABELNUMBER,LABELS,LABELREF);
362      << ASSIGNS LINE NUMBERS TO ALL BRANCH STATEMENTS USING LABELS >>
363      INTEGER LABELNUMBER;
364      INTEGER ARRAY LABELREF;
365      BYTE ARRAY LABELS;
366      BEGIN
367      DOUBLE LOOP2;
368          FOR LOOP:=1 UNTIL LINENUMBER DO
369          BEGIN
370              LOOP2:=DOUBLE(LOOP);
371              FREADDIR(OBJECTNUMBER,CODE,3,LOOP2);
372              IF < THEN MESSAGE;
373              IF CODE(1) = 6 THEN
374              BEGIN
375                  CODE(2):=LABELREF(CODE(2));
376                  IF CODE(2) = 0 THEN ERROR:=16;
377                  FWRITEDIR(OBJECTNUMBER,CODE,3,LOOP2);
378                  IF < THEN MESSAGE
379              END;
380          END;
381      END;
382
383      INTEGER PROCEDURE LOAD;
384      << LOADS THE PROPER DATA INTO THE OPERAND ACCORDING TO THE
385         ADDRESSING TECHNIQUE >>
386      BEGIN
387          IF CODE(1) = 0 THEN
388              LOAD:=OPERAND
389          ELSE
390              LOAD:=STACK(OPERAND)
391      END;
392
393      PROCEDURE BRANCH;
394      << RESETS THE PROGRAM COUNTER AND CHECKS FOR ERRORS >>
395      BEGIN
396          IF CODE(1) = 6 THEN
397              COUNTER:=OPERAND - 1
398          ELSE
399              COUNTER:=COUNTER - (OPERAND + 1);
```

```
400         IF (COUNTER < 0) OR (COUNTER > LINENUMBER) THEN
401             ERROR:=17;
402     END;
403
404     PROCEDURE ROUND(NUMBER);
405     INTEGER NUMBER;
406     BEGIN
407         IF (NUMBER > 255) OR (NUMBER < -255) THEN
408         BEGIN
409             NUMBER:=NUMBER.(8:8);
410             IF OPERAND = 0 THEN STATUS.(2:1):=STATUS.(2:1) * (-1)
411         END;
412     END;
413
414     PROCEDURE RUN;
415     << EXECUTES THE OBJECT CODE STORED IN TEMP FILE "OBJECT" >>
416     BEGIN
417     INTRINSIC CAUSEBREAK;
418     DOUBLE LOOP2;
419         COUNTER:=1;
420         STATUS:=0;
421         STACKPOINTER:=100;
422         EOF:=FALSE;
423         LOOP2:=DOUBLE(COUNTER);
424         FREADDIR(OBJECTNUMBER,CODE,3,LOOP2);
425         IF > THEN EOF:=TRUE;
426         IF < THEN MESSAGE;
427         WHILE NOT EOF DO
428         BEGIN
429             CASE CODE(1) OF BEGIN
430                 OPERAND:=CODE(2);
431                 OPERAND:=CODE(2);
432                 OPERAND:=CODE(2) + XREG;
433                 OPERAND:=CODE(2) + YREG;
434                 OPERAND:=STACK(CODE(2));
435                 OPERAND:=0;
436                 OPERAND:=CODE(2);
437             END;
438             CASE CODE(0) OF BEGIN
439                 ;
440                 CAUSEBREAK;                 <<1>>
441                 ;                           <<2>>
442                 STATUS.(2:1):=0;            <<3>>
443                 BEGIN                       <<4>>
444                     XREG:=XREG - 1;
445                     OPERAND:=1;
446                     ROUND(XREG)
447                 END;
448                 BEGIN                       <<5>>
449                     YREG:=YREG - 1;
450                     OPERAND:=1;
451                     ROUND(YREG)
452                 END;
453                 BEGIN                       <<6>>
454                     XREG:=XREG + 1;
455                     OPERAND:=1;
456                     ROUND(XREG)
```

```
457                 END;
458                 BEGIN                          <<7>>
459                     YREG:=YREG + 1;
460                     OPERAND:=1;
461                     ROUND(YREG)
462                 END;
463                 ;                              <<8>>
464                 BEGIN                          <<9>>
465                     STACK(STACKPOINTER):=STACK(0);
466                     STACKPOINTER:=STACKPOINTER - 1
467                 END;
468                 BEGIN                          <<10>>
469                     STACK(STACKPOINTER):=STATUS;
470                     STACKPOINTER:=STACKPOINTER -1
471                 END;
472                 BEGIN                          <<11>>
473                     STACKPOINTER:=STACKPOINTER + 1;
474                     STACK(0):=STACK(STACKPOINTER)
475                 END;
476                 BEGIN                          <<12>>
477                     STACKPOINTER:=STACKPOINTER + 1;
478                     STATUS:=STACK(STACKPOINTER)
479                 END;
480                 BEGIN                          <<13>>
481                     STACKPOINTER:=STACKPOINTER + 1;
482                     COUNTER:=STACK(STACKPOINTER)
483                 END;
484                 STATUS.(2:1):=1;               <<14>>
485                 XREG:=STACK(0);                <<15>>
486                 YREG:=STACK(0);                <<16>>
487                 XREG:=STACKPOINTER+1;          <<17>>
488                 STACK(0):=XREG;                <<18>>
489                 BEGIN                          <<19>>
490                    STACKPOINTER:=XREG;
491                    STACKPOINTER:=STACKPOINTER - 1
492                 END;
493                 STACK(0):=YREG;                <<20>>
494                 RETURN;                        <<21>>
495                 ;;;;;;;;                      <<22-29>>
496                 ;                               <<30>>
497                 ;                               <<31>>
498                 IF STACK(0) = 0 THEN           <<32>>
499                   BRANCH;
500                 IF STACK(0) < 0 THEN           <<33>>
501                   BRANCH;
502                 IF STACK(0) <> 0 THEN          <<34>>
503                   BRANCH;
504                 IF STACK(0) > 0 THEN           <<35>>
505                   BRANCH;
506                 IF STATUS.(2:1)=0 THEN<<36>>
507                   BRANCH;
508                 IF STATUS.(2:1)=1 THEN<<37>>
509                   BRANCH;
510                 ;;                             <<38-39>>
511                 BEGIN                          <<40>>
512                     COUNTER:=OPERAND - 1;
513                     IF (COUNTER > LINENUMBER) OR (COUNTER < 1) THEN
```

```
514                         ERROR:=17
515                 END;
516                 BEGIN                           <<41>>
517                     STACK(STACKPOINTER):=COUNTER;
518                     STACKPOINTER:=STACKPOINTER + 1;
519                     COUNTER:=OPERAND;
520                     IF (COUNTER > LINENUMBER) OR (COUNTER < 0) THEN
521                         ERROR:=17
522                 END;
523                 ;;;;;;;;                        <<42-49>>
524                 BEGIN                           <<50>>
525                     XREG:=LOAD;
526                     ROUND(XREG)
527                 END;
528                 BEGIN                           <<51>>
529                     YREG:=LOAD;
530                     ROUND(YREG)
531                 END;
532                 BEGIN                           <<52>>
533                     IF (OPERAND < 0) OR (OPERAND > OPLIMIT) THEN
534                         ERROR:=17
535                     ELSE
536                         STACK(OPERAND):=XREG
537                 END;
538                BEGIN                            <<53>>
539                    IF (OPERAND < 0) OR (OPERAND > OPLIMIT) THEN
540                        ERROR:=17
541                    ELSE
542                        STACK(OPERAND):=YREG
543                END;
544                 ;;;;;;                          <<54-59>>
545                 BEGIN                           <<60>>
546                     IF (OPERAND < 0) OR (OPERAND > OPLIMIT) THEN
547                         ERROR:=18
548                     ELSE BEGIN
549                         STACK(OPERAND):=STACK(OPERAND) * 2;
550                         ROUND(STACK(OPERAND))
551                     END;
552                 END;
553                 BEGIN                           <<61>>
554                     IF (OPERAND < 0) OR (OPERAND > OPLIMIT) THEN
555                         ERROR:=18
556                     ELSE BEGIN
557                         STACK(OPERAND):=STACK(OPERAND) + 1;
558                         ROUND(STACK(OPERAND))
559                     END;
560                 END;
561                 STACK(OPERAND):=STACK(OPERAND)/2;<<62>>
562                 BEGIN                           <<63>>
563                     IF (OPERAND < 0) OR (OPERAND > OPLIMIT) THEN
564                         ERROR:=18
565                     ELSE BEGIN
566                         STACK(OPERAND):=STACK(OPERAND) - 1;
567                         ROUND(STACK(OPERAND))
568                     END;
569                 END;
570                 BEGIN                           <<64>>
```

```
571                     IF (OPERAND < 0) OR (OPERAND > OPLIMIT) THEN
572                         ERROR:=17
573                     ELSE
574                         STACK(OPERAND):=STACK(0)
575                 END;
576                 BEGIN                           <<65>>
577                     LENGTH:=ASCII(LOAD,10,OPBYTE);
578                     LENGTH:=LENGTH * (-1);
579                     IF STACK(OPLIMIT+1) = 0 THEN
580                         PRINT(OPBYTE,LENGTH,%320);
581                     IF STACK(OPLIMIT+1) = 1 THEN
582                         PRINT(OPBYTE,LENGTH," ");
583                     IF STACK(OPLIMIT+1) = 2 THEN
584                         PRINT(OPBYTE,LENGTH,"0");
585                     IF STACK(OPLIMIT+1) = 3 THEN
586                         PRINT(OPBYTE,LENGTH,"1");
587                     IF STACK(OPLIMIT+1) > 3 THEN
588                     BEGIN
589                         WARNING:=3;
590                         PRINTWARNING(WARNING,WARNINGS);
591                         PRINT(OPBYTE,LENGTH," ")
592                     END;
593                 END;
594                 BEGIN                           <<66>>
595                     LENGTH:=READ(OPBYTE,-4);
596                     IF (OPERAND < 0) OR (OPERAND > OPLIMIT) THEN
597                         ERROR:=18
598                     ELSE BEGIN
599                         STACK(OPERAND):=BINARY(OPBYTE,LENGTH);
600                         ROUND(STACK(OPERAND))
601                     END;
602                 END;
603                 BEGIN                               <<67>>
604                     OPBYTE(0):=BYTE(LOAD);
605                     IF STACK(OPLIMIT+1) = 0 THEN
606                         PRINT(OPBYTE(0),-1,%320);
607                     IF STACK(OPLIMIT+1) = 1 THEN
608                         PRINT(OPBYTE(0),-1," ");
609                     IF STACK(OPLIMIT+1) = 2 THEN
610                         PRINT(OPBYTE(0),-1,"0");
611                     IF STACK(OPLIMIT+1) = 3 THEN
612                         PRINT(OPBYTE(0),-1,"1");
613                     IF STACK(OPLIMIT+1) > 3 THEN
614                     BEGIN
615                         WARNING:=3;
616                         PRINTWARNING(WARNING,WARNINGS);
617                         PRINT(OPBYTE(0),-1," ");
618                     END;
619                 END;
620                 BEGIN                           <<68>>
621                     READ(OPBYTE,-1);
622                     IF (OPERAND < 0) OR (OPERAND > OPLIMIT) THEN
623                         ERROR:=18
624                     ELSE BEGIN
625                         STACK(OPERAND):=INTEGER(OPBYTE(0));
626                         ROUND(STACK(OPERAND))
627                     END;
```

```
628                 END;
629                 ;                           <<69>>
630                 BEGIN                       <<70>>
631                     STACK(0):=LOAD+STACK(0);
632                     ROUND(STACK(0))
633                 END;
634                 BEGIN                       <<71>>
635                     STACK(0):=INTEGER(LOGICAL(STACK(0)) LAND
636                         LOGICAL(LOAD));
637                     ROUND(STACK(0))
638                 END;
639                 ;                           <<72>>
640                 BEGIN                       <<73>>
641                     STACK(0):=LOAD;
642                     ROUND(STACK(0))
643                 END;
644                 BEGIN
645                     STACK(0):=INTEGER(LOGICAL(STACK(0)) LOR <<74>>
646                         LOGICAL(LOAD));
647                     ROUND(STACK(0))
648                 END;
649                 BEGIN                       <<75>>
650                     STACK(0):=STACK(0)-LOAD;
651                     ROUND(STACK(0))
652                 END;
653                 STACK(OPLIMIT+1):=LOAD;<<76>>
654             END;
655             IF STACK(0) < 0 THEN STATUS.(0:1):=1
656                 ELSE STATUS.(0:1):=0;
657             IF STACK(0) = 0 THEN STATUS.(1:1):=1
658                 ELSE STATUS.(1:1):=0;
659             IF STACK(0).(8:1) = 1 THEN STATUS.(5:1):=1
660                 ELSE STATUS.(5:1):=0;
661             COUNTER:=COUNTER + 1;
662             LOOP2:=DOUBLE(COUNTER);
663             FREADDIR(OBJECTNUMBER,CODE,3,LOOP2);
664             IF > THEN EOF:=TRUE;
665             IF < THEN MESSAGE;
666             IF ERROR > 0 THEN
667             BEGIN
668                 PRINTERROR(ERROR,ERRORS);
669                 MOVE DISPLAYLINE(0):=80(" ");
670                 PRINT(DISPLAYLINE,-80," ");
671                 MOVE DISPLAYLINE(0):=
672                     "PROGRAM TERMINATED IN AN ERROR STATE";
673                 PRINT(DISPLAYLINE,-80,"0");
674                 EOF:=TRUE
675             END;
676         END;
677     END;
678
679     << ********************* >>
680     << BEGINNING OF PROGRAM >>
681     << ********************* >>
682
683         LINENUMBER:=0;
684         LABELNUMBER:=0;
```

```
685           VARIABLENUMBER:=0;
686           STACKPOINTER:=OPLIMIT;
687           STACK(OPLIMIT+1):=1;
688           SOURCENUMBER:=FOPEN(SOURCEFILE,%5,%1300,-72);
689           IF < THEN MESSAGE;
690           OBJECTNUMBER:=FOPEN(OBJECTFILE,%0,%5,3);
691           IF < THEN MESSAGE;
692           FREAD(SOURCENUMBER,LINE,-72);
693           IF > THEN EOF:=TRUE;
694           IF < THEN MESSAGE;
695           MOVE DISPLAYLINE:="6502 ASSEMBLY LANGUAGE SIMULATOR- V1 ";
696           DATELINE(DISPLAYLINE(37));
697           PRINT(DISPLAYLINE,-80,"0");
698           MOVE DISPLAYLINE:=80(" ");
699           PRINT(DISPLAYLINE,-80,"0");
700       WHILE NOT EOF DO
701           BEGIN
702               LINENUMBER:=LINENUMBER+1;
703               PARSER(LINENUMBER,LINE,LABELS,LABELNUMBER,OPERATOR,
704                   TECHNIQUE,OPBYTE,OPLABEL,OPVARIABLE,VARIABLES,
705                   ERROR,WARNING);
706               CODE(0):=OPCODE(OPERATOR);
707               MOVE DISPLAYLINE:=80(" ");
708               IF (CODE(0) <> 80) AND (CODE(0) <> 81) THEN
709                   ASCII(LINENUMBER,10,DISPLAYLINE(0));
710               FOR LOOP:=0 UNTIL 71 DO
711                   DISPLAYLINE(9+LOOP):=LINE(LOOP);
712               LENGTH:=(-1)*(CURSOR+9);
713               PRINT(DISPLAYLINE,LENGTH," ");
714               IF ERROR <> 0  THEN  PRINTERROR(ERROR,ERRORS);
715               IF WARNING <> 0 THEN PRINTWARNING(WARNING,WARNINGS);
716               IF CODE(0) = 0 THEN
717               BEGIN
718                   ERROR:=3;
719                   PRINTERROR(ERROR,ERRORS)
720               END;
721               IF CODE(0) = 80 THEN ADDLABEL(LABELS,LABELNUMBER,OPLABEL);
722               IF CODE(0) = 81 THEN
723                   ADDVARIABLE(VARIABLES,VARIABLENUMBER,OPVARIABLE);
724               ADDRESSING:=-1;
725               IF TECHNIQUE = "#" THEN ADDRESSING:=0;
726               IF TECHNIQUE = "$" THEN ADDRESSING:=1;
727               IF TECHNIQUE = "X" THEN ADDRESSING:=2;
728               IF TECHNIQUE = "Y" THEN ADDRESSING:=3;
729               IF TECHNIQUE = "V" THEN ADDRESSING:=1;
730               IF TECHNIQUE = "L" THEN ADDRESSING:=6;
731               IF TECHNIQUE = "I" THEN ADDRESSING:=4;
732               IF TECHNIQUE = "P" THEN ADDRESSING:=4;
733               IF TECHNIQUE = "-" THEN ADDRESSING:=5;
734               CODE(1):=ADDRESSING;
735               IF (CODE(0)<25) AND (CODE(0)<>0) AND (TECHNIQUE<>"-") THEN
736                   ERROR:=10;
737               IF (CODE(0)>=30) AND (CODE(0)<40) AND (TECHNIQUE<>"L")
738                    AND (TECHNIQUE<>"#") THEN ERROR:=11;
739               IF (CODE(0)>=40) AND (CODE(0)<50) AND (TECHNIQUE<>"L")
740                    AND (TECHNIQUE<>"#") THEN ERROR:=12;
741               IF (CODE(0)>=50) AND (CODE(0)<60) AND (TECHNIQUE<>"#")
```

```
742                 AND (TECHNIQUE<>"$") AND (TECHNIQUE<>"V") AND
743                 (TECHNIQUE<>"I") AND (TECHNIQUE<>"P") THEN ERROR:=13;
744               IF (CODE(0)>=60) AND (CODE(0)<70) AND (TECHNIQUE<>"$")
745                 AND (TECHNIQUE<>"X") AND (TECHNIQUE<>"Y") AND
746                 (TECHNIQUE<>"V") AND (TECHNIQUE<>"I") AND
747                 (TECHNIQUE<>"P") THEN ERROR:=13;
748               IF (CODE(0)>=70) AND (TECHNIQUE<>"#") AND (TECHNIQUE<>
749                 "$") AND (TECHNIQUE<>"X") AND (TECHNIQUE<>"Y") AND
750                 (TECHNIQUE<>"V") AND (TECHNIQUE<>"I") AND
751                 (TECHNIQUE<>"P") AND (CODE(0)<80)THEN ERROR:=13;
752               IF (CODE(0)=80) AND (TECHNIQUE <> "L") THEN
753                 ERROR:=13;
754               IF (CODE(0)=81) AND (TECHNIQUE <> "V") THEN
755                 ERROR:=13;
756               IF ERROR >0 THEN PRINTERROR(ERROR,ERRORS);
757
758               IF (TECHNIQUE="#") OR (TECHNIQUE="$") OR (TECHNIQUE="X")
759                 OR (TECHNIQUE="Y") OR (TECHNIQUE="I") THEN
760                 CODE(2):=BINARY(OPBYTE,4);
761               IF (TECHNIQUE="V") OR (TECHNIQUE="P") THEN
762               BEGIN
763                 CODE(2):=-1;
764                 FOR LOOP:=0 STEP 10 UNTIL VARIABLENUMBER DO
765                 BEGIN
766                   TESTVARIABLE:= TRUE;
767                   FOR LOOP1:=0 UNTIL 9 DO
768                     IF VARIABLES(LOOP+LOOP1)<>OPVARIABLE(LOOP1) THEN
769                       TESTVARIABLE:=FALSE;
770                   IF TESTVARIABLE THEN
771                     CODE(2):=LOOP/10+1
772                 END;
773                 IF CODE(2) = -1 THEN
774                 BEGIN
775                   ERROR:=4;
776                   PRINTERROR(ERROR,ERRORS)
777                 END;
778               END;
779               IF TECHNIQUE = "L" THEN
780               BEGIN
781                 CODE(2):=-1;
782                 FOR LOOP:=0 STEP 10 UNTIL LABELNUMBER DO
783                 BEGIN
784                   TESTLABEL:= TRUE;
785                   FOR LOOP1:=0 UNTIL 9 DO
786                   IF LABELS(LOOP+LOOP1) <> OPLABEL(LOOP1) THEN
787                     TESTLABEL:= FALSE;
788                   IF TESTLABEL THEN
789                     CODE(2):=LOOP/10
790                 END;
791                 IF CODE(2)=-1 THEN
792                 BEGIN
793                   ERROR:=5;
794                   PRINTERROR(ERROR,ERRORS)
795                 END;
796               END;
797               IF (CODE(0) <> 80) AND (CODE(0) <> 81) THEN
798               BEGIN
```

```
799                 DOUBLENUMBER:=DOUBLE(LINENUMBER);
800                 FWRITEDIR(OBJECTNUMBER,CODE,3,DOUBLENUMBER);
801                 IF < THEN MESSAGE;
802             END
803             ELSE LINENUMBER:=LINENUMBER - 1;
804             FREAD(SOURCENUMBER,LINE,-72);
805             IF > THEN EOF:= TRUE;
806             IF < THEN MESSAGE;
807         END;
808     << CONTINUE WITH RUN OF PROGRAM IF NO ERRORS >>
809         MOVE DISPLAYLINE:=80(" ");
810         PRINT(DISPLAYLINE,-80,"0");
811         MOVE DISPLAYLINE(0):="NUMBER OF ERRORS:";
812         ASCII(ERRORS,10,DISPLAYLINE(19));
813         MOVE DISPLAYLINE(25):="NUMBER OF WARNINGS:";
814         ASCII(WARNINGS,10,DISPLAYLINE(45));
815         PRINT(DISPLAYLINE,-80,"0");
816         IF ERRORS = 0 THEN
817         BEGIN
818             MOVE DISPLAYLINE:=80(" ");
819             MOVE DISPLAYLINE:="END OF COMPILE STEP";
820             PRINT(DISPLAYLINE,-80,"0");
821             PREPARE(LABELNUMBER,LABELS,LABELREF);
822             IF ERROR = 0 THEN
823             BEGIN
824                 MOVE DISPLAYLINE:="END OF PREPARE STEP";
825                 PRINT(DISPLAYLINE,-80,"0");
826                 RUN
827             END
828             ELSE BEGIN
829                 PRINTERROR(ERROR,ERRORS);
830                 MOVE DISPLAYLINE:="PROGRAM TERMINATED IN AN ERROR STATE";
831                 PRINT(DISPLAYLINE,-80,"0")
832             END;
833         END
834         ELSE BEGIN
835             MOVE DISPLAYLINE:=80(" ");
836             MOVE DISPLAYLINE:="PROGRAM TERMINATED IN AN ERROR STATE";
837             PRINT(DISPLAYLINE,-80,"0")
838         END;
839     END.
```

APPENDIX E: SUPPLIMENTARY MATERIAL FOR USE WITH THIS SYSTEM

THE NATURE OF THE SYSTEM--COMPILER, ASSEMBLER, OR SIMULATOR?

When electronic computers were first manufactured, the programmer had only the most primitive methods to program it. All programming was done through binary codes, a switch-flipping type process which was slow and tedious. It shortly became apparent that there must be a better way.

A better way came with the advent of assemblers. These programs took other programs written in an "assembly language" code and translated them into a machine language that the computer could understand. Assemblers made the task of programming much less tedious, but the language was still machine-dependent--that is, one assembly language command executed only one machine instruction. In order to realize the full power of the computer, a language had to be developed that would execute several machine tasks with one command, keep track of tables and arrays, and generally free the programmer to concentrate on the finer points of programming.

These higher level languages (among the most notable are BASIC, FORTRAN, COBOL, RPG, AND PASCAL) were developed with this goal in mind. The programs used to break these languages down into machine language are called compilers. Compilers are usually written in assembly language.

Unfortunately, computer science is noted for exceptions, and Hewlett-Packard fits the bill. They developed a language, Systems Programming Language (SPL), which was designed to be high-level language, yet still machine-oriented. SPL would then be used to write compilers, supervisors, and utility programs. The assembly language is still present, in a sense, but it is buried beneath other options designed to increase the power and useability of the language.

SPL is an effective and powerful language, but it is inappropriate to use as a learning tool in a beginning assembly language course. The student must grasp the basic concepts of assembly language itself before trying to handle the intricacies of SPL. It was with this problem in mind that I set out to develop a simpler assembly language to use on the Hewlett-Packard.

Ironically, I found SPL to be the most suitable language to program the system. In order to write a compiler/assembler for a language that was machine oriented, I had to program it in a language that was also machine oriented. Also, SPL is a fast-executing language, desirable for a system which is already burdened with excessive demands from interactive users.

While Hewlett-Packard has its own machine instruction set, it has no real assembly language of its own, therefore I had to choose another well-known assembly language to

simulate on the HP machine. I chose a language based on the 6502 microprocessor. This is a relatively simple language to learn and many students would already be familiar with it through existing microprocessor courses.

This simulator acts like an assembler, but is actually written in SPL, a high-level language. This process is "transparent" to the user, meaning that in all respects, the system looks and acts like an assembler, when in actuality it is only simulating one. In all appearances, the student is using assembly language.

THE SOURCE CODE

In SPL, all procedures must come before the main part of the program, so the program actually starts at line 683. The program executes as follows:

INITIALIZATION (LINES 683-699):

Assigns starting values, opens the input and object files and prints the heading.

PARSER (LINES 119-251):

Breaks each string of characters in a line of code down into its component parts. This procedure uses another procedure called OPERATOR (lines 55-117), which assigns a code to the parsed operator.

SOURCE LISTING (LINES 706-720):

Prints the source listing plus any error or warning messages resulting from syntax errors detected by the parser. PRINTERROR (lines 253-282) and PRINTWARNING (lines 284-299) are the two procedures used with this block of code.

LABEL ADDITION (LINES 314-335)

Adds any labels to the label table as specified by a LAB command.

VARIABLE ADDITION (LINES 337-359)

Adds any labels to the label table as specified by a VAR command.

ADDRESSING TECHNIQUE CHECK (LINES 724-796)

Checks to make sure the proper operand is used with the addressing technique, and assigns numerical values to variables and labels.

OBJECT FILE GENERATION (LINES 797-8070

Generates the "object" file to be used in the execution phase.

PREPARATION FOR EXECUTION (LINES 361-381)

Modifies the object file to replace label numbers with the actual line numbers in branch statements.

EXECUTION (LINES 414-617)

Executes the object code. Three procedures are used with this block of code--LOAD (lines 383-319), which loads a value dependent on the addressing technique, BRANCH (lines 393-402), which handles conditional and unconditional branching, and ROUND (lines 404-412) which handles data overflows and sets the overflow flag.

VARIABLE LIST

The following is a list of all variables used by the source code:

INTEGERS-

LINENUMBER - the program counter

LABELNUMBER - first empty position in the label array

VARIABLENUMBER - first empty position in the variable array

ERROR - current error number

ERRORS - total number of errors

WARNING - current warning number

WARNINGS - total number of warnings

SOURCENUMBER - input file number

OBJECTNUMBER - "object" file number

LOOP, LOOP1, LOOP2 - used for loop counters

STACKPOINTER - program stack pointer

XREG - X-register

YREG - Y-register

STATUS - status register

COUNTER - current line number during execution

OPERAND - current operand during execution

CURSOR, LAST - substring pointers used for parsing

LENGTH - length of input/output strings

ADDRESSING - object code addressing technique

DOUBLENUMBER - double word number needed for certain functions

ARRAYS OF INTEGERS

STACK - program stack

CODE - "object" code

LABELREF - label reference table

ARRAYS OF CHARACTERS

SOURCEFILE - input file name

OBJECTFILE - "object" file name

LINE - unparsed source line

LABELS - label list

VARIABLES - variable list

OPERATOR - parsed operator

OPBYTE - parsed operand if an ASCII digit

OPLABEL - parsed operand if a statement label

OPVARIABLE - parsed operand if a variable

TECHNIQUE - parsed addressing technique

DISPLAYLINE - output line buffer

LOGICAL VARIABLES-

EOF - end-of-file flag

TESTVARIABLE - variable test flag

TESTLABEL - label test flag

INCREASING MEMORY CAPABILITY

The system was originally set up with a stack limit of 100 and space for 20 variables and 20 labels. This may be expanded by making small changes to the source code and re-compiling it.

Line 25 defines the upper limit of the stack. The line can be changed to reflect the new upper limit.

Line 28 defines the stack array itself, including the carriage control character. The stack should be redefined so that there will be one byte available for the carriage control character. For instance, if the stack limit defined in line 25 is 200, the stack itself should have a lower limit of 0 and an upper limit of 201. Position 0 would be the accumulator, positions 1-200 would be the program stack, and position 201 would be the carriage control character.

Lines 37-38 define the label and variable stack. The upper limit is determined by multiplying the maximum number of variables or labels by 10, then subtracting 1. For instance, if the maximum number of labels desired is 50, then the upper limit would be 50 x 10 - 1 = 499.

Line 30 defines the label cross-reference table. Its upper limit should be redefined to be one less than the maximum number of labels desired.

APPENDIX F: SAMPLE PROGRAMS

```
:JOB SAFIGAN,PHYSICS,SAFIGAN
PRIORITY = DS; INPRI = 8; TIME = UNLIMITED SECONDS
JOB NUMBER = #J735
WED, MAR 30, 1983,  5:43 PM                                                    56
HP3000 / MPE IV C.A0.20
:FILE SOURCE=SOURCE4
:RUN EUREKA

6502 ASSEMBLY LANGUAGE SIMULATOR- V1 WED, MAR 30, 1983,  5:43 PM


          LAB L START
1         LDA #0045
2         STA $0005
3         LDX #0000
4         START: TXA -0000
5         ADC #0001
6         CCT #0000
7         PRT $0000
8         PRA $0005
9         PRA $0005
10        PRA $0005
11        PRA $0005
12        CCT #0001
13        PRA $0000
14        TAX -0000
15        ADC #0128
16        BNE L START
17        STP -0000
18        NOP - 0000
END OF FILE  (FSERR 0)

NUMBER OF ERRORS:  0      NUMBER OF WARNINGS: 0

END OF COMPILE STEP

END OF PREPARE STEP

1----
2----
3----
4----
5----
6----
7----
8----
9----
10----
11----
12----
13----
14----
15----
16----
17----
18----
19----
20----
21----
22----
23----
24----
25----
26----
27----
28----
29----
30----
31----
32----
33----!
34----"
35----#
36----$
37----%
38----&
39----'
40----(
```

41----)
42----*
43----+
44----,
45----
46----.
47----/
48----0
49----1
50----2
51----3
52----4
53----5
54----6
55----7
56----8
57----9
58----:
59----;
60----<
61----=
62---->
63----?
64----@
65----A
66----B
67----C
68----D
69----E
70----F
71----G
72----H
73----I
74----J
75----K
76----L
77----M
78----N
79----O
80----P
81----Q
82----R
83----S
84----T
85----U
86----V
87----W
88----X
89----Y
90----Z
91----[
92----\
93----]
94----^
95----
96----`
97----a
98----b
99----c
100----d
101----e
102----f
103----g
104----h
105----i
106----j
107----k
108----l
109----m
110----n
111----o
112----p
113----q
114----r
115----s
116----t
117----u
118----v
119----w
120----x

```
121----y
122----z
123----{
124----|
125----}
126----~
127----
128----

END OF PROGRAM
:EOJ
CPU SEC. = 11.  ELAPSED MIN. = 1.  WED, MAR 30, 1983,  5:44 PM
```

```
:JOB CARL.COBOL,ASSEMBLY
PRIORITY = DS; INPRI = 8; TIME = UNLIMITED SECONDS
JOB NUMBER = #J736
WED, MAR 30, 1983,  5:49 PM
HP3000 / MPE IV C.A0.20
:ASSEMBLEGO MPLY3
                                                                        59
6502 ASSEMBLY LANGUAGE SIMULATOR- V1 WED, MAR 30, 1983,  5:50 PM


                                    LAB L BEGIN
                                    LAB L END
                                    LAB L MULTIPLY_LOOP
                                    LAB L ADD
                                    LAB L NO_PRODUCT_OVERFLOW
                                    LAB L DON'T_ADD
                                    LAB L NO_MPLICAND_OVERFLOW
                                    LAB L WRITE_ERROR
                                    VAR V MPLICAND
                                    VAR V MPLIER
                                    VAR V PRODUCT
1                         BEGIN:    CCT # 0000
2                                   LDA # 0201
3                                   PRA $ 0000
4                                   LDA # 0238
5                                   PRA $ 0000
6                                   LDA # 0240
7                                   PRA $ 0000
8                                   LDA # 0245
9                                   PRA $ 0000
10                                  LDA # 0244
11                                  PRA $ 0000
12                                  LDA # 0032
13                                  PRA $ 0000
14                                  LDA # 0244
15                                  PRA $ 0000
16                                  LDA # 0119
17                                  PRA $ 0000
18                                  LDA # 0111
19                                  PRA $ 0000
20                                  LDA # 0032
21                                  PRA $ 0000
22                                  LDA # 0238
23                                  PRA $ 0000
24                                  LDA # 0117
25                                  PRA $ 0000
26                                  LDA # 0109
27                                  PRA $ 0000
28                                  LDA # 0098
29                                  PRA $ 0000
30                                  LDA # 0101
31                                  PRA $ 0000
32                                  LDA # 0114
33                                  PRA $ 0000
34                                  LDA # 0115
35                                  CCT # 0001
36                                  PRA $ 0000
37                                  INP V MPLICAND
38                                  INP V MPLIER
39                                  LDA # 0000
40                                  STA V PRODUCT
41                  MULTIPLY_LOOP:  LDA # 0001
42                                  AND V MPLIER
43                                  BEQ L DON'T_ADD
44                              ADD:LDA V PRODUCT
45                                  ADC V MPLICAND
46                                  BVS L WRITE_ERROR
47              NO_PRODUCT_OVERFLOW:STA V PRODUCT
48                        DON'T_ADD:LSR V MPLIER
49                                  LDA # 0255
50                                  SBC V PRODUCT
51                                  SBC V MPLICAND
52                                  SBC V MPLICAND
53                                  BVS L WRITE_ERROR
54              NO_MPLICAND_OVERFLOW:ASL V MPLICAND
55                                  LDA V MPLIER
56                                  BNE L MULTIPLY_LOOP
57                                  PRT V PRODUCT
58                                  JMP L END
```

```
59                          WRITE_ERROR:  LDA  #  0101
60                                        PRA  $  0000
61                                        LDA  #  0114
62                                        PRA  $  0000
63                                        LDA  #  0114
64                                        PRA  $  0000
65                                        LDA  #  0111
66                                        PRA  $  0000
67                                        LDA  #  0114
68                                        PRA  $  0000
69                                  END:  STP  -  0000
70                                        NOP  -  0000
END OF FILE  (FSERR 0)

NUMBER OF ERRORS:  0      NUMBER OF WARNINGS: 0

END OF COMPILE STEP

END OF PREPARE STEP

Input two numbers 12,13
156

END OF PROGRAM
:EOJ
CPU SEC. = 5.  ELAPSED MIN. = 1.  WED, MAR 30, 1983,  5:50 PM
```

Steve Safigan attended Mississippi College from 1979 to 1983, where he majored in mathematics and computer science. He currently works with Southern Farm Bureau Life Insurance Company as an Actuarial Student.